



Static and Dynamic Software Security Analysis

Project report

Submitted By

Musfika Ikfat Munia

ID: 32774211

Ojo Olumuyiwa

ID:

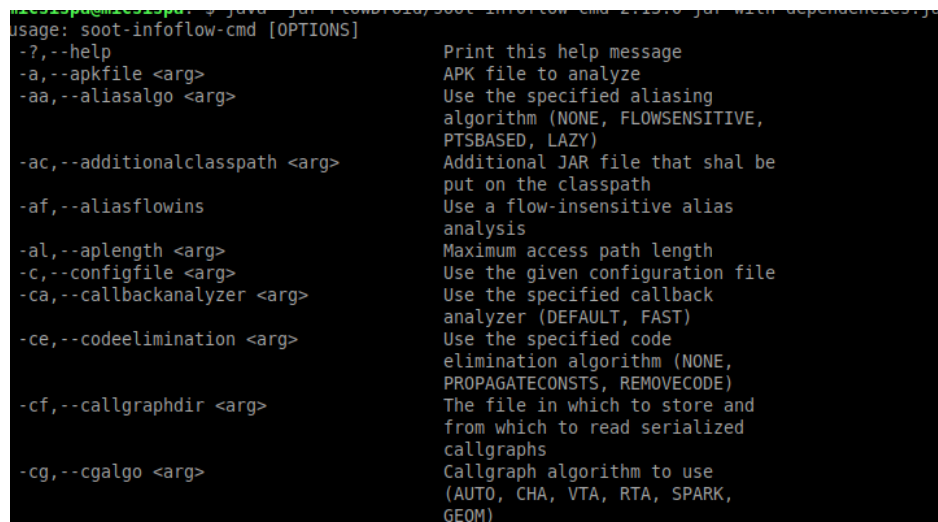
Section 1: Introduction to FlowDroid for Taint Analysis

Android app components are the essential building blocks of an Android application, each serving a distinct purpose in the app's lifecycle and user interaction. The four main components include Activities (managing the user interface), Services (handling background tasks), Broadcast Receivers (responding to system-wide events), and Content Providers (managing shared app data). These components, defined in the app's manifest file, create dynamic and interactive user experiences.

In this report, we have analyzed 10 APKs that were provided and were analyzed using FlowDroid for potential data leaks.

Understand FlowDroid and Configuration Options

FlowDroid is a widely used static taint analysis tool designed for Android applications. It helps identify how sensitive data ("taints") flows through an application by tracing its sources and sinks. FlowDroid reports data flows between sources and sinks. Some of the configuration settings are presented in Figure 1.

A screenshot of a terminal window showing the usage and options for the 'soot-infoflow-cmd' tool. The text is as follows:

```
usage: soot-infoflow-cmd [OPTIONS]
-?,--help                               Print this help message
-a,--apkfile <arg>                     APK file to analyze
-aa,--aliasalgo <arg>                  Use the specified aliasing
                                         algorithm (NONE, FLOWSENSITIVE,
                                         PTSBASED, LAZY)
-ac,--additionalclasspath <arg>        Additional JAR file that shall be
                                         put on the classpath
-af,--aliasflowins                       Use a flow-insensitive alias
                                         analysis
-al,--aplength <arg>                   Maximum access path length
-c,--configfile <arg>                  Use the given configuration file
-ca,--callbackanalyzer <arg>           Use the specified callback
                                         analyzer (DEFAULT, FAST)
-ce,--codeelimination <arg>            Use the specified code
                                         elimination algorithm (NONE,
                                         PROPAGATECONSTS, REMOVECODE)
-cf,--callgraphdir <arg>               The file in which to store and
                                         from which to read serialized
                                         callgraphs
-cg,--cgalgo <arg>                     Callgraph algorithm to use
                                         (AUTO, CHA, VTA, RTA, SPARK,
                                         GEOM)
```

Figure 1.

Identify Sources and Sinks

Source: The entry point in the code where sensitive data enters the application. Examples include:

- User input, such as data entered in forms or text fields.
- Device information, such as location, IMEI, contacts, or photos.
- API calls that retrieve sensitive data, such as `LocationManager.getLastKnownLocation()`.

Sink: Locations in the code where data is handled in a way that could pose a security risk. Examples include:

- Network transmissions, such as HTTP POST requests using `URLConnection` or libraries like `OkHttp`.
- Writing data to files, such as external storage or logs.
- Logging data through system logs, such as `Log.d` or `Log.e` in Android.
- Sending information via SMS or email.

Run FlowDroid Analysis

In order to analyze the provided apps, FlowDroid was installed through the following link: [FlowDroid GitHub](#).

Figure 2 displays the command to run the data flow tracker on each APK file.

```

micsispa@micsispa:~$ java -jar FlowDroid/soot-infoflow-cmd-2.13.0-jar-with-dependencies.jar -a Downloads/APKs/com.wooxhome.smart.apk -p Android/Sd
k/platforms/ -s FlowDroid/soot-infoflow-android/SourcesAndSinks.txt --timeout 1200 > wooxhomeSmart20.txt 2>&1
micsispa@micsispa:~$ ls
Android  Desktop  Downloads  eclipse  eclipse-workspace  firefox  git  Pictures  sootOutput  Templates  Videos  wooxhomeSmart20.txt
Desktop  Desktop  Downloads  eclipse  eclipse-workspace  firefox  git  Pictures  sootOutput  Templates  Videos  wooxhomeSmart1.txt  wooxhomeSmart5.txt

```

Figure 2

- a** references the apk file
- p** references the Android JAR folder as the "platforms" directory inside the Android SDK installation folder. android-33 was used for the project.
- s** references the SourceandSink file used for the project with the labels Source&Sink. The default file provided in the repository was used for the project
- timeout** stops the data flow analysis after the time set. It is set in seconds.
- resulttimeout** stops the path reconstruction after the time set. This was used when the data flow analysis of a few apps took a considerable amount of time to complete.
- 2>&1**: Redirects error output (stderr) to the same file as standard output (stdout). When running tools like FlowDroid, errors (stderr) may contain important diagnostic information (e.g., warnings, missing files). Without **2>&1** the error messages will not be captured in the output file

We can observe the output of the command in Figure 2 saved as text files.

Each APK has been analyzed in three periods: 1 minute (60 sec), 5 minutes (300 sec), and 20 minutes (1200 sec).

Document Hardware and Configuration:

The analysis was conducted using the VM provided for the course and configured as follows:

- **Operating System:** Ubuntu 64-bit
- **Allocated Resources:**
 - **RAM:** 4096 MB (4 GB)
 - **CPU:** 3 cores allocated with a CPU usage cap of 100%
 - **Disk Space:** 20 GB Virtual Disk (SATA interface)
 - **Graphics:** Controller: VMSVGA, Video Memory: 16 MB
 - **Networking:** Intel PRO/1000 MT Desktop adapter (NAT mode)

Analyze and Discuss Results

All the 10 APKs were analyzed with different timeouts and the results can be observed in Table 1 below.

Nº	App	Number of leaks (60 sec)	Number of leaks (300 sec)	Number of leaks (1200 sec)
1	com.delhi.metro.dtc	3	3	3
2	com.hawaiianairlines.app	59	59	59
3	com.imo.android.imoim	4	4	4
4	com.tado	15	15	15
5	com.walkme.azores.new	5	5	5

№	App	Number of leaks (60 sec)	Number of leaks (300 sec)	Number of leaks (1200 sec)
6	com.wooxhome.smart	0	0	0
7	com.yourdelivery.pyszne	7	7	7
8	linko.home	24	24	24
9	mynt.app	16	16	16
10	nz.co.stuff.android.news	26	26	26

Table 1 – Analysis results

Due to the volume of leaks reported from each app analysis in Table 1, we have selected a potential data leak from each app.

The potential data leaks are described below;

com.delhi.metro.dtc:

- **Source:**
\$d0 = virtualinvoke \$r6.<android.location.Location: double getLatitude()>. \$d0 = virtualinvoke \$r6.<android.location.Location: double getLongitude()> - captures location data (getLatitude() and getLongitude()).
- **Sink**
staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>("sandeep", \$r8) - data is being logged using android.util.Log.

Logging location data (latitude and longitude) is a serious concern. Even if used for debugging, exposing such data via logs is risky, especially if the app interacts with external systems that can access logs. This is a genuine data leak.

com.hawaiianairlines.app:

- **Source:** virtualinvoke \$r8.<android.net.wifi.WifiInfo: java.lang.String getSSID()> - sensitive data: Wi-Fi SSID.
- **Sink:** virtualinvoke \$r0.<androidx.activity.ComponentActivity: void startActivityForResult(android.content.Intent,int)> - found in multiple activities such as MediaActivity, FavoritesActivity, and ShoppingActivity.

Passing Wi-Fi SSID between activities might indicate functional requirements such as pairing or contextual actions. However, this practice risks exposing sensitive data if the Intent is not securely handled (e.g., broadcasting).

com.imo.android.imoim:

- **Source:** virtualinvoke \$r8.<android.content.pm.PackageManager: java.util.List queryBroadcastReceivers(android.content.Intent,int)> - information queried from the PackageManager, potentially related to broadcast receivers.

- **Sink:** staticinvoke <android.util.Log: int i(java.lang.String,java.lang.String)>("AppsFlyer_6.11.0", \$r2) - data is logged using android.util.Log.

Querying broadcast receivers and logging the results can be a debugging practice but risks exposing sensitive app-related information in production. It can be a potential data leak, as app-level data should not be logged in production.

com.tado:

- **Source:** virtualinvoke \$r9.<java.net.HttpURLConnection: java.io.InputStream getInputStream()> - reads data from an HTTP connection's input stream.
- **Sink:** staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String) > ("AccountResponse", \$r1) - logs sensitive data as part of API response debugging.

Debugging API responses by logging input streams risks exposing sensitive data, especially in production environments. It may be considered as a genuine data leak.

- **Source:** virtualinvoke r0.<com.tado.android.installation.CreateAccountActivity: android.view.View findViewById(int)> - extracts UI elements during account creation.
- **Sink:** virtualinvoke \$r12.<java.io.OutputStream: void write(byte[])> - writes data to an output stream.

Writing UI data to an output stream might be expected during data submission (e.g., forms). Validation of data use is required. It is unlikely to be a data leak, assuming proper usage.

com.walkme.azores.new:

- **Source:** virtualinvoke \$r4.<java.util.Locale: java.lang.String getCountry()> - retrieves the user's country code from the system locale.
- **Sink:** interfaceinvoke \$r7.<android.content.SharedPreferences\$Editor: android.content.SharedPreferences\$Editor putString(java.lang.String,java.lang.String)> - stores the country code in shared preferences.

Saving the user's locale (e.g., country code) is a common functionality. However, the security of shared preferences must be evaluated. If preferences are secured, it is unlikely to be a data leak.

linko.home:

- **Source:** \$r9 = virtualinvoke \$r1.<java.io.File: java.io.File getAbsolutePath()> - accesses the absolute file path.
- **Sink:** virtualinvoke \$r16.<java.io.FileOutputStream: void write(byte[])> - writes the file to an output stream.

Writing absolute file paths to output streams may expose sensitive file system data (potential data leak) if improperly handled.

- **Source:** \$r10 = interfaceinvoke \$r13.<android.database.Cursor: java.lang.String getString(int)> - retrieves data from a database cursor.
- **Sink:** virtualinvoke \$r6.<android.os.Bundle: void putString(java.lang.String,java.lang.String)> - stores sensitive data in a Bundle.

Storing database query results in a Bundle without encryption can lead to privacy concerns (genuine data leak).

com.mynt.app:

- **Source:** \$r21 = virtualinvoke \$r19.<android.content.pm.PackageManager: java.util.List nz.co.stuff.android.news:queryIntentServices(android.content.Intent,int)> - retrieves information about services from the package manager.
- **Sink:** staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)> - logs sensitive service query results.

Logging service query results may expose app-specific behaviors and configurations, posing a security risk (genuine data leak).

nz.co.stuff.android.news:

- **Source:** \$r7 = interfaceinvoke \$r6.<android.database.Cursor: java.lang.String getString(int)> - retrieves data from a database cursor.
- **Sink:** virtualinvoke \$r3.<java.io.Writer: void write(java.lang.String,int,int)>(\$r2, 0, \$i0) - writes the data into a file.

Writing sensitive data retrieved from a database to a file might align with app functionality but requires strong data protection measures. It can be a potential data leak, depending on the file storage mechanism's security.

Evaluate the Impact of Timeout

The analysis measures the number of leaks detected for each app under three different timeout settings: **60 seconds**, **300 seconds**, and **1200 seconds**.

The number of leaks remains constant for all analyzed apps regardless of the timeout setting. The constant leak counts across all timeout settings suggest that FlowDroid completes each app's analysis within the minimum timeout (60 seconds). Increasing the timeout does not uncover additional leaks.

It can be inferred from the result presented in Table 1 that a timeout of 60 seconds appears sufficient for detecting leaks in all the analyzed apps.

Extending the timeout to 300 or 1200 seconds does not benefit these specific APKs.

[Optional] Repeat with Different Hardware Setup

The same procedure was executed in a Windows environment with the following hardware specifications to replicate the above method.

- **Operating System:** Windows 11, 64-bit
- **Allocated Resources:**
 - **RAM:** 16534 MB (15.8 GB usable)
 - **CPU:** Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz 1.90 GHz
 - **Disk Space:** 512 GB SSD

Figure 2

Figure 2 - Implementation

*

№	App	Personal PC	№	App	Personal PC
1	com.delhi.metro.dtc	3	6	com.wooxhome.smart	0
2	com.hawaiianairlines.app	65	7	com.yourdelivery.pyszne	7
3	com.imo.android.imoim	4	8	linko.home	23
4	com.tado	15	9	mynt.app	16
5	com.walkme.azores.new	5	10	nz.co.stuff.android.new.s	26

Table 2 – Optional task analysis results (1200 sec)

```
PS C:\Users\ADMIN\Desktop\UL\SAST&DAST\Project> java -jar .\FlowDroid\soot-infoflow-cmd-2.13.0-jar-with-dependencies.jar -a .\APKs\linko.home.apk -p C:\Users\ADMIN\AppData\Local\Android\Sdk\platforms\ -s .\FlowDroid\soot-infoflow-android\SourcesAndSinks.txt --timeout 1200 --resulttimeout 1200 > Link20.txt 2>&1
PS C:\Users\ADMIN\Desktop\UL\SAST&DAST\Project> java -jar .\FlowDroid\soot-infoflow-cmd-2.13.0-jar-with-dependencies.jar -a .\APKs\com.delhi.metro.dtc.apk -p C:\Users\ADMIN\AppData\Local\Android\Sdk\platforms\ -s .\FlowDroid\soot-infoflow-android\SourcesAndSinks.txt --timeout 1200 --resulttimeout 1200 > Delhi20PC.txt 2>&1
PS C:\Users\ADMIN\Desktop\UL\SAST&DAST\Project> java -jar .\FlowDroid\soot-infoflow-cmd-2.13.0-jar-with-dependencies.jar -a .\APKs\com.walkme.azores.new.apk -p C:\Users\ADMIN\AppData\Local\Android\Sdk\platforms\ -s .\FlowDroid\soot-infoflow-android\SourcesAndSinks.txt --timeout 1200 --resulttimeout 1200 > walkme20PC.txt 2>&1
```

Figure 3. Command Execution on Windows Environment

Section 2: Results and Insights Regarding the Two Privacy Requirements

R1:

To verify compliance with R1, we used FlowDroid, a static taint analysis tool, to analyze the com.yourdelivery.pyszne.apk food ordering application. The strategy involved the following steps:

Selection of Sources and Sinks

- **Sources:** We identified methods that retrieve location data as sources. These include methods from the *android.location.Location* and *android.location.LocationManager* classes.
- **Sinks:** We identified methods that send data over the network as sinks. These include methods from the *java.net* and *org.apache.http* packages.

Configuration of FlowDroid

- A new source and sink file was created to detect if any location-related information was sent outside the app..

```
micsispa@micsispa:~$ java -jar FlowDroid/soot-infoflow-cmd-2.13.0-jar-with-dependencies.jar -a Downloads/APKs/com.yourdelivery.pyszne.apk -p Android/Sdk/platforms/ -s FlowDroid/soot-infoflow-android/SourcesSinksLocation.txt --timeout 1200 --resulttimeout 1200 > R1Y.txt 2>&1
micsispa@micsispa:~$ nano R1Y.txt
```

Figure 4.

Running the Analysis

Results of the Analysis for R1

```

[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - IFOS problem with 2101115 forward and 1675022 backward edges solved in 48 seconds, processing 2 r
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Current memory consumption: 948 MB
[main] INFO soot.jimple.infoflow.memory.FlowDroidTimeoutWatcher - FlowDroid timeout watcher started
[main] INFO soot.jimple.infoflow.memory.FlowDroidTimeoutWatcher - FlowDroid timeout watcher terminated
[main] INFO soot.jimple.infoflow.data.pathBuilders.DefaultPathBuilderFactoryShutdownBatchPathBuilder - Running path reconstruction batch 1 with 2 elements
[main] INFO soot.jimple.infoflow.data.pathBuilders.DefaultPathBuilderFactoryRepeatableContextSensitivePathBuilder - Obtained 2 connections between sources and sinks
[main] INFO soot.jimple.infoflow.data.pathBuilders.DefaultPathBuilderFactoryRepeatableContextSensitivePathBuilder - Building path 1...
[main] INFO soot.jimple.infoflow.data.pathBuilders.DefaultPathBuilderFactoryRepeatableContextSensitivePathBuilder - Building path 2...
[main] INFO soot.jimple.infoflow.memory.MemoryWarningSystem - Shutting down the memory warning system...
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Memory consumption after path building: 414 MB
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Path reconstruction took 1 seconds
[main] INFO soot.jimple.infoflow.memory.FlowDroidTimeoutWatcher - FlowDroid timeout watcher terminated
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - The sink staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>($r5, $r1) in m
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - $r0 = virtualinvoke $r1.<android.location.Location: double getLatitude()>() in method <com.yopeso
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - $r0 = virtualinvoke $r1.<android.location.Location: double getLongitude()>() in method <com.yopeso
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - $r0 = virtualinvoke $r1.<android.location.Location: double getLatitude()>() in method <com.yopeso
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - $r4 = virtualinvoke $r4.<android.view.View: android.view.View findViewById(int)>($i0) in method
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - $r4 = virtualinvoke $r4.<android.view.View: android.view.View findViewById(int)>($i0) in method
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Data flow solver took 53 seconds. Maximum memory consumption: 948 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication - Found 2 leaks

```

Figure 5. R1 Output

The analysis found two potential data leak in the com.yourdelivery.pyszne.apk application. The details of the data leak are as follows:

Identified Potential Data Leak

Sink: <android.util.Log: int d(java.lang.String,java.lang.String)>

Source: <android.location.Location: double getLatitude()>

Description:

- The latitude from the Location object is logged, potentially exposing sensitive location data.
- This occurs in the method <com.yopeso.lieferando.fragments.search.CSHFragment\$1: void onLocationChanged(android.location.Location)>.

Sink: <android.util.Log: int d(java.lang.String,java.lang.String)>

Source: <android.location.Location: double getLongitude()>

Description:

- The longitude from the Location object is logged, resulting in possible leakage of sensitive information.
- This occurs in the method <com.yopeso.lieferando.fragments.tablet.search.SearchMainFragmentTablet\$1: void onLocationChanged(android.location.Location)>.

Privacy Requirements

Logging sensitive data such as location is considered **data leakage**.

Even though the logged data is not explicitly sent over the network, it becomes accessible through system logs. Malicious apps with access to system logs could extract this information.

Challenges Encountered

The major challenge was creating a new source and sink file to verify compliance with R1. It took an iterative process of identifying sources and sinks.

R2:

Verify whether the user can delete their account. We employed the use of different approaches.

1. As with R1, we created a new source and sink text file to verify compliance by running it with FlowDroid. However, it yielded no possible leaks, as indicated in Figure 6 below..

```

[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Starting (aint Analysis
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Using context- and flow-sensitive solver
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Using context- and flow-sensitive solver
[main] INFO soot.jimple.infoflow.memory.FlowDroidTimeoutWatcher - FlowDroid timeout watcher started
[main] WARN soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Running with limited join point abstractions can break context-sensitive path builders
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Looking for sources and sinks...
[main] ERROR soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - No sinks found, aborting analysis
[main] INFO soot.jimple.infoflow.memory.MemoryWarningSystem - Shutting down the memory warning system...
[main] WARN soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - No results found.
[main] INFO soot.jimple.infoflow.android.SetupApplicationsInPlaceInfoflow - Data flow solver took 3 seconds. Maximum memory consumption: 319 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication - Found 0 leaks

```


2. Due to the result from 1, we explored further by decompiling the APK through jadx and reviewing the AndroidManifest.xml to see if there are any permission sets like Manage_Accounts or Authenticate_Accounts. However, we noticed an implementation of AccountActivity.
3. Further exploration of the AccountActivity class, it handles functionalities on Account details, address management, orders, logout, etc. We did not locate the account deletion functionality.

```
/* loaded from: classes.dex */
public class AccountActivity extends BaseAccountActivity implements View.OnClickListener {
    public static final String ADD_ADDRESS_ACTION = "add_address_action";
    private static final String PHOTO_FRAGMENT = "PhotoFragment";
    private Bundle bundle;
    private WebViews fragment;
    private MapAddressFragment mapAddressFragment;
}
```

Figure 7. Account Activity Class

4. We also installed the application on Android Studio to get a feel for the UI. However, the application got stuck and could not load the interface.



Figure 8. App Interface

Challenges Encountered

Also, with R2, the challenge was creating a new source and sink file to verify compliance with R2. In addition, getting the app installed took a reasonable amount of time as its targeted version has been deprecated. See Figure 9.

```
<uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="21"/>
```

Figure 9.

Privacy Requirements

After the app analysis, we could not detect the account deletion's functionality despite implementing the account management activity. Hence, the app is not GDPR compliant.

Conclusion.

The hands-on experience of the project, primarily through the use of the FlowDroid analysis tool, helped to put into proper context the challenges faced when performing analysis on Android applications.

Also, some of the leaks discovered when put into the proper context of the application were not considered data leaks. This opens up the conversation around reducing false positives.